CHAPTER 6

Programming Multithreaded Applications

In This Chapter

- Understanding the basic uses for threads
- Understanding the types of threads that MFC supports
- Determining which type of thread will work best for your application
- Keeping threads from conflicting by using critical sections
- Learning how to create, suspend, resume, terminate, and schedule a thread
- Maintaining efficient thread execution speed by setting its priority
- Working with threads within applications
- Working with threads using DLLs
- Creating libraries that enable safe thread execution

One of the things that makes working in the Windows 32-bit environment different from the 16-bit environment is having the ability to perform more than one task at a time within a single application. This ability is provided through the use of threads. A thread is essentially a single subject of execution within an application. In other words, a single application can do more than one task at a time. For example, it can print and spell check a document in the background while the user is typing in the foreground.

You need to be aware of two entities when talking about threads. The term *thread* describes one set of contiguous instructions for performing a single task. A process, on the other hand, describes the application as a whole. Every executing application has at least one thread and one process. There's never more than one process for an application, but Win32 applications can always produce more than one thread. The distinction between processes and threads will become clearer as the chapter progresses.

Threads don't perform magic. They won't make the user's machine any faster, and the processor still can't perform more than one task at a time. In other words, threads don't enable a machine to simultaneously perform more than one task unless that machine has the resources (that is, multiple processors) to do so.

What threads do allow is more efficient use of machine resources and better response times for the user. A document can print in the background while the user works in the foreground because the user isn't likely to require full use of the processor all of the time. Making use of idle time while the

user thinks enables the machine to work more efficiently. Because a particular task gets taken care of only during idle time, the user regains use of the machine immediately after requesting that the task get performed.

This chapter provides you with a wealth of information about how threads work and how you can use them in your applications to make things run more efficiently. It's important to understand how threads get used in various types of project types, as well as the kinds of threads you have at your disposal when using Visual C++ and MFC. (There isn't any actual distinction between threads from a pure Win32 perspective.)

We'll also explore safety considerations to think about when using threads. For example, critical sections ensure that two calling processes don't try to use the same portion of code at the same time. However, even if you have your code properly written to ensure thread safety, the libraries that you use may not be written with the same considerations in mind. In this chapter, I also explore the things you need to consider to ensure thread safety when working with local libraries.

Finally, we'll spend time working through two example programs. The first program is designed to show you how to use threads within an application. The second program will deal with using threads within libraries. Obviously, part of this example's purpose is to show the library in action so that you can actually see the effect of using threads in this environment.

Uses for Threads

Theoretically, you can use threads in any kind of Visual C++ application, including something as small as an ActiveX component. Threads also don't necessarily need to be large or complex to make an application more efficient and responsive to user needs. In fact, you can use threads to perform small maintenance tasks in the background at regular intervals — something you may not want to interrupt your main application to do. A thread can also replace timer-related tasks in some cases. In other words, threads aren't limited to performing any particular task.

However, there are some things you need to consider before you start using threads for every small task that your application may need to perform. The following list provides you with some guidelines on what you should consider before using threads.

Debugging

The biggest consideration from a developer perspective is that threads greatly increase the difficulty of debugging an application. A thread can actually hide bugs, or at least make them more difficult to find since you now have to watch more than one thread of execution at a time.

Development	Most developers are used to thinking about application programming in a fairly linear fashion. In other words, given a specific event, the application will perform a series of steps to handle it. Using multiple threads forces the programming to think about the application processes in a parallel rather than a linear fashion.
True Efficiency	Though it's true that placing some tasks into background threads can make use of idle time in the application, there are situations in which there isn't any idle time to exploit. In such a situation, you'll find that the application is actually less efficient than before because a certain amount of overhead and housekeeping is associated with using multiple threads. In other words, only use threads in situations in which you anticipate some amount of idle time to exploit.
Reliability	Multiple threads of execution don't necessarily make an application failure prone, but there are more failure points to consider. Any time you add more failure points to anything, it becomes less reliable. There's a greater probability that the application will break simply because more things can go wrong with it.
Unexpected Side Effects	No matter how carefully you craft a multithreaded application, there will be side effects that you have to deal with, especially if the threads in the application interact. Even if you make your application thread safe and use critical sections, there's a chance that two threads will try to access the same variable at the same time in an unanticipated way. Not only do these unexpected side effects increase development and debugging time, they make it more likely that a user will come across a problem that you can't duplicate with your setup. In other words, multithreaded applications will more than likely increase application support costs.

Now that you have a good overview of the way in which threads can be used in general, let's look at some specific multithreaded usage types. In the following sections, we will explore the three most common ways that you'll see multiple threads in use: applications, DLLs, and system services. Each of these areas represents a major application type. We'll explore two of these multithreaded usage examples, applications and DLLs, later in the chapter.

Applications

We've already explored this topic to some extent. Applications can benefit from multiple threads of execution in a number of ways. In fact, some of those ways will seem quite natural from a programming perspective because the tasks in question can be broken from the main thread of execution quite easily. The following sections give you some ideas on how you can use multiple threads with applications.

Printing

This is the one major task that can always benefit from multiple threads in any application. Queuing a print job takes time, which means that the user is sitting at his or her desk, staring at the screen, doing nothing at all. In fact, some print jobs take so much time that the user will give up trying to use the computer at all and do something else while waiting. Printing in the background in a separate thread is always an efficient way to handle this task.

There are probably a few things you should consider not adding to background threads simply because it's not feasible to do so. The one rule of thumb you should go by is whether the user will need to interact directly with the thread. In many cases, anything that requires direct interaction on a constant basis should be handled as part of the main thread. On the other hand, anything the user can set once, then allow the computer to complete, is a good candidate for a separate thread. Make sure the application will realize an efficiency gain and the user increased application responsiveness any time you create a thread.

As the User Types

A lot of tasks fall into the "as the user types" category, but the two most common are spelling and grammar checks. A lot of applications offer the ability to check the user's spelling and grammar as they type, which reduces the need to check the whole document later. Of course, a lot of less common tasks fall into this category as well. For example, you could check the validity of an equation as the user types it or make sure that a database entry is correct. For that matter, you could even suggest (as some applications do) a completed entry for the user based on past input.

Repetition

Repagination and other repetitive tasks can always occur as background threads. There isn't any need to take up the foreground task's time with things like updating the application clock. Most repetitive, continuous tasks can be relegated to a background thread.

Data Saves

Most applications now include an automatic save feature simply because many users are very poor at saving data themselves. It's not hard to figure out why — the user is engrossed in getting the document completed and simply forgets to perform

the data save. An automatic data saving feature can enable the user to complete a document without worrying about power failures or other computer glitches that can cause data to disappear.

Updates

As users rely more and more on remote computing, the need to provide them with updates in the field increases. Updates, in this case, aren't necessarily limited to data. For example, a user might check in with the company each morning for updated pricing schedules. A system administrator could use this habit to his or her advantage by including a background thread that downloads any system updates the user may require. In other words, the user would receive both a data update and an application update at the same time. Of course, automatic data updates are a nice feature as well. The application could update pricing tables or other forms of application-specific information in the background at regular intervals provided the machine has the capability of creating a remote connection to the company.

You can combine multiple threads and system updates in other ways. For example, you might want to include a virus-checking thread that runs in the background and checks all of the incoming data before it actually gets placed on the client machine. Another use of background threads includes running diagnostics in the background, as the user works, to ensure his or her machine is fully functional. An alarm would tell the user that the machine requires service and that he or she should save any data before it's too late. As you can see, you can use threads in a lot of ways to protect the user, the data, and the client machine from damage.

Calculations

Math operations are notorious for consuming vast amounts of processor cycles. In some cases, you have to accept the heavy penalty of a calculation because the user requires an answer immediately. However, there are other situations in which the calculation could be completed just as easily in the background as a separate thread. In fact, many spreadsheet and graphics applications now use this technique to make foreground features more responsive to user input.

DLLs

Dynamic link libraries (DLLs) have been around since Windows was first created. In fact, DLLs are actually the predecessors of the libraries used by DOS applications. For the most part, DLLs allow for the same uses of threads as applications do. The main difference is that you would want to place common thread types in DLLs — threads that perform work that you may need to do in more than one application. However, some thread categories do get placed in DLLs simply because they are major components of an application that the developer may not want to recompile every time the application is updated.

System Services

For the most part, users never interact with system services. System services sit in the background and do things like enabling a machine's hardware to operate or creating network connections. As a result, there are some fairly specialized uses for threads within a system service. I will discuss some of these possible uses in the following sections.

Service Priority Upgrade

Some system services are low-priority background tasks to begin with. You normally don't want them to consume valuable processor cycles unless the machine is idle or there is some type of priority task to perform. It's when you use a service in this second capacity that high-priority threads come into play. Rather than change the priority of the entire service, you can simply launch a single thread to perform the high-priority task.

Discovery

Most system services are low-level applications that need to discover a great deal about the system to ensure that it's working properly. This discovery phase can occur once during service initialization in some cases, but in other cases, it's an ongoing process. Consider the network driver that has to keep track of the current system configuration, including the status of remote resources. A good use of threads, in this case, would be to enable the service to perform multiple levels of discovery at the same time, without reducing its availability to the system as a whole.

An Overview of Thread Types

As far as Windows is concerned, there are threads and the processes that contain them, and nothing else. However, from an MFC perspective, there are actually two kinds of threads: UI and worker. Both are threads that can perform a single sequence of execution within the application. The difference comes in the way that these two kinds of threads are implemented and used. In the following sections, I talk about these two thread types and how they're used.

You can use the Win32 `CreateThread()` function to create a thread that doesn't rely on MFC. The advantage of doing so is that you eliminate some of the overhead normally encountered using the MFC libraries. In addition, this method conserves memory. The downside, of course, is that you can't use any of the capabilities that MFC provides. In most cases, you'll find that `CreateThread()` works best for worker threads that perform simple repetitive tasks.

Worker Threads

Worker threads are normally used for background tasks that require no or minimal user interaction. They're implemented as a function that returns a UINT result and accepts one argument of the LPVOID data type as shown here:

```
UNINT MyThread (LPVOID pParam)
{
    return 0;
}
```

A worker thread normally returns a value of 0, which indicates that it successfully completed whatever task it was designed to perform. You can return other values to indicate either errors or usage counts. However, the calling application has to be designed to retrieve the exit value using the `GetExitCodeThread()` function.

Another way to end a worker thread and generate an exit code is to use the `AfxEndThread()` function. Using this function will stop thread execution and perform any required cleanup prior to exiting to the calling application. The calling application would still need to use the `GetExitCodeThread()` function to retrieve the exit value provided to the `AfxEndThread()` function. The exact meaning of any exit codes is up to you, so the calling application will need to be designed to work with a specific thread function before it will know what an exit code means.

The `pParam` argument can contain any number of 32-bit values. However, passing a pointer to a data structure has several benefits that you may want to consider. For one thing, using a structure allows you to pass more than one argument to the thread. In many cases, a single argument won't be enough to provide the thread with everything needed to perform useful work, so a structure is the only way to get around the single-input argument requirement. In addition, using a structure allows the worker thread to pass information back to the caller. The only thing that the worker thread would need to do is modify the contents of a structure member during the course of execution.

UI Threads

As the name suggests, UI threads are normally created to provide some type of user-interface functionality within an application. You'll derive the UI thread from the CWinThread class instead of using a function, as with the worker thread. Obviously, this means that implementing a UI thread is more complex than a worker thread, but you also get more flexibility.

Terminating a UI thread is much the same as terminating a worker thread. However, a UI thread requires a little special handling if you want the caller to retrieve the exit code for the thread. First, you need to set the m_bAutoDelete data member to FALSE, which prevents the CWinThread object from deleting itself. Second, you'll need to manually delete the thread and release any memory that it uses. As an alternative, you can always duplicate the CWinThread handle that you receive during thread creation using the `DuplicateHandle()` method. In this case, you'll want to create the thread in the suspended state, duplicate the handle, then start the thread using the `ResumeThread()` method.

There's only one CWinThread class method that you must override when creating a new UI thread, though several others are commonly overridden as well. The InitIntance( ) method is the one method that you must override because it is the first one called after the thread is created. The InitInstance( ) method should contain all of the code required to initialized your thread. Obviously, this means displaying a main dialog for the thread, if necessary.

The ExitInstance( ) method will normally get overridden only if you need to perform some thread cleanup or postprocessing. The only place that you can call this method from is the Run( ) method (should you decide to override it as well). ExitInstance( ) performs the default tasks of deleting the CWinThread object, if m_bAutoDelete is TRUE. It's always the last method called before the thread terminates.

Of the other methods available to you, the only other methods that you may need to override are OnIdle( ), Run( ), PreTranslateMessage( ), and ProcessWndProcException( ). The OnIdle( ) method handles any idle time processing for the thread. For example, OnIdle( ) would get called if the application displayed a dialog box and the user wasn't doing anything with it. Run( ) controls the flow of activity within the thread, including the message pump. PreTranslateMessage( ) filters messages before they are sent to either TranslateMessage( ) or DispatchMessage. Finally, the ProcessWndProcException( ) method handles any unhandled exceptions thrown by the thread's message and command handlers. However, you'd normally want to handle these exceptions within the handler rather than wait until it reaches this point of the thread.

Understanding Critical Sections

A critical section is a piece of code that can be accessed by only one thread of an application at one time. In addition, some forms of critical section ensure that the code sequence will get completed without interruption. You may want to create a critical section for a lot of reasons; the most important of which is data integrity. An application changes the contents of variables and the status of objects to meet the needs of a particular user. If another user suddenly decides to execute the same code, then the contents of those variables and objects would be ruined for both parties.

You can create a critical section for use with threads in two ways: a variable of type CRITICAL_SECTION or a CCriticalSection object. Of the two, the CCriticalSection object is the easier to use and the least error prone. All you need to do is create a CCriticalSection object, then use the Lock( ) and Unlock( ) methods as needed to either restrict or allow access to a section of code. Here's an example of what a typical critical section sequence might look like:

```
CCriticalSection    oMySection;    // Critical section object.

// Lock the critical section.
if (!oMySection.Lock())
    AfxMessageBox("Failed to lock critical section", MB_OK);

// Do some critical work here.

// Unlock the critical section.
if (!oMySection.Unlock())
    AfxMessageBox("Failed to unload critical section.", MB_OK);
```

Using the CCriticalSection object is easy. It returns a BOOL value that signifies success or failure. You'll normally need to handle a failure to either lock or unlock the critical section in some way. In this case, I used a simple dialog. A production application would likely use some kind of loop to try to lock the critical section multiple times. Note that you can supply a numeric value for the Lock( ) method that tells how long to wait for the critical section to get unlocked before a failure result gets returned. Make sure you set this value to a reasonable amount so that the user doesn't think the application is frozen.

Ensuring Libraries Are Thread Safe

One of the benefits of using libraries is code reuse. Once the code is written and tested, you can place it in a library and forget about it. The functionality you'll need will be available without a lot of additional work.

Unfortunately, the black box functionality of libraries can be a two-edged sword. One of the biggest problems when using libraries with threads is that the library isn't thread safe. In other words, if two threads attempt to access the same object or function at the same time, there could be a collision, resulting in a frozen application, lost data, or other unanticipated results.

Fortunately, you can protect your libraries in a number of ways. One way is to use critical sections as needed to ensure that a sequence of events takes place without interruption. A second way is to allocate variables and objects on the stack. Finally, it's extremely important to reduce the risk of collisions by not allowing more than one thread to access the same variable—use techniques that ensure that each thread will have its own set of variables to use.

Even Microsoft's libraries aren't totally thread safe. For the most part, any MFC object is thread safe at the class level, but not at the object level. In other words, two threads could access different objects derived from the same class, but not the same object. If you have a single CString object that two different threads need to access, then you need to write code that ensures that each object will have separate access to the CString object. In addition, the first thread will need to completely finish any work with the CString object before you allow access by the second thread.

If you want to use MFC at all within a thread, you must create that thread using one of the two techniques we discussed earlier. These are the only two methods that ensure you can access the MFC libraries safely. The reason you must use these recommended techniques is that they tell the MFC libraries to initialize internal variables that allow for multithreaded access of the routines.

Writing a Basic Application

As mentioned in the previous paragraphs, you can use threads in a lot of different ways in an application and MFC provides two different kinds of threads to choose from. This example is going to look at both kinds of threads in a very simple scenario. We'll use dialog boxes just to make the whole interface of the project that much easier.

CD-ROM Location: Chapt06\Threads
Program Name: Threads.EXE
Source Code Modules in Text: NewThreadDlg.H, NewThreadDlg.CPP, Threads.RC, ThreadsDlg.H, ThreadsDlg.CPP, UIThread.H, and UIThread.CPP

The first part of the example will use a UI thread. You'll normally call on this kind of thread when you need to create multiple windows to display information for the user. For example, you might use this kind of thread setup when writing a network diagnostic program. A single window could display all of the statistics for one problem point on the network. The network administrator could then look at the windows one at a time and prioritize the most severe ones for immediate repair.

The second part of the example will look at a worker thread. The UI thread will actually call the worker thread as part of the initialization process. Worker threads are normally used to perform nonuser interface tasks in the background. In this case, our worker thread won't really do all that much. All that it'll do is wait five seconds and terminate.

Defining the Main Dialog

Now that you have an idea of where this project is going, let's create the main dialog for it. This will include creating the application itself. The following procedure will help you get the project set up.

The " A Simple Dialog-based Application" section of Chapter 3 provides the necessary familiarity with the basic procedures that you'll need to understand the following example. Make sure you spend some time learning the basic application types before proceeding to the more complex examples in the book.

Try This

1. Create a new MFC AppWizard (exe) project named Threads. You'll see the MFC AppWizard - Step 1 dialog.
2. Choose the Dialog-based option, then click Finish. You'll see a New Project Information dialog like the one shown in Figure 6-1.

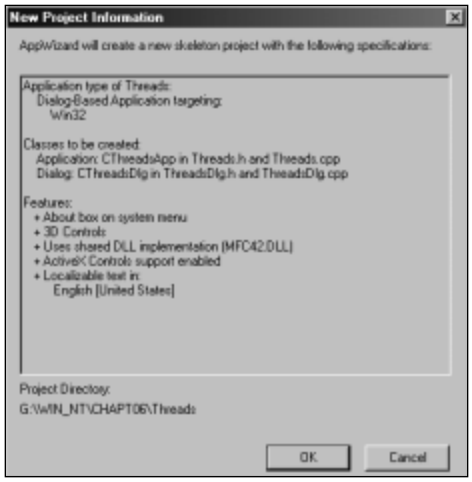

Figure 6-1: The New Project Information dialog for the Threads project

3. Click OK. Visual C++ will create the new project for you, then display the main application dialog in the Design window.
4. Remove the Static Text control and add a Button control.
5. Open the Push Button Properties dialog. Change the ID to IDC_ADD_THREAD and the Caption to Add Thread. Your dialog should look like the one shown in Figure 6-2.

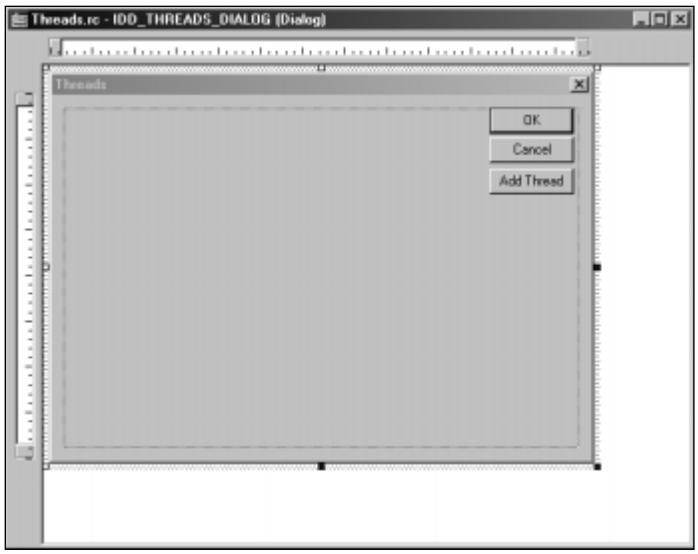

Figure 6-2: The IDD_THREADS_DIALOG resource will be the main application dialog.

6. Ctrl+Double-click IDC_ADD_THREAD. Visual C++ will add a new procedure for the button to the application.

At this point, the main application dialog is ready to go. All we need to do is add some code to make the Add Thread button functional. Listing 6-1 shows the code you'll need to add.

Listing 6-1: **The OnOK Event Handler Source Code**

```
void CThreadsDlg::OnAddThread()
{
    // Begin a new UI thread.
    CWinThread    *pThread = AfxBeginThread(
        RUNTIME_CLASS ( CUIThread ) );
}
```

As you can see, this code is extremely simple. All we do is create a new thread based on the CUIThread class. Unfortunately, there is no CUIThread class associated with our application at the moment. That's the next step — creating a new class to hold our thread.

Creating the CUIThread Class

The first kind of thread we'll create is a UI thread. What this thread will do is display a dialog box when the user clicks the Add Thread button. The following procedure will show you how to create the class.

Try This

1. Use the Insert | New Class command to display the New Class dialog shown in Figure 6-3. This dialog will allow you to create any kind of new class, but we'll use it to create a new thread for our application.

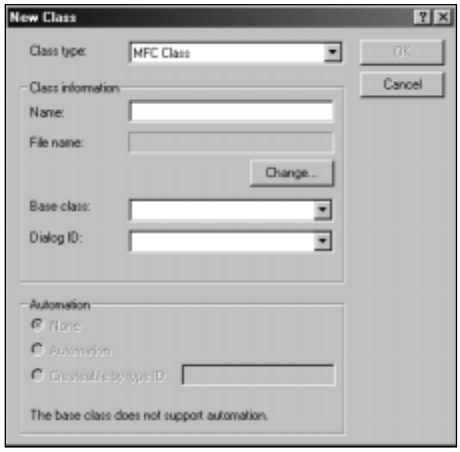

Figure 6-3: The New Class dialog allows you to add new classes to your application.

2. Type CUIThread in the Name field. (Visual C++ will automatically name the files for the new class UIThread — you can change this default filename by clicking Change.) The value you provide in the Name field determines the class name throughout the application. It's traditional to preface all class names with a C, though there is no rule that you absolutely have to.
3. Choose CWinThread in the Base Class field. The base class determines the initial characteristics of the class. Some classes, like CDialog, require you to provide a Dialog ID as well, but we don't need one in this case.
4. Click OK. Visual C++ will add the CUIThread class to the application.

Even though we've added the CUIThread thread class to the application, the CThreadsDlg class, where we added the OnAddThread( ) method in the previous section, still can't see it. We need to add an include statement at the beginning of the ThreadsDlg.CPP like the one shown here:

```
// Add our thread header.
#include "UIThread.h"
```

At this point, you could compile and run the application. If you clicked the Add Thread button, you wouldn't see anything happen with the application itself. However, if you looked at the application with the Process Viewer utility that's supplied with Visual C++, you'd see one additional thread for each time the Add Thread button was pressed. Figure 6-4 shows what you might see if you'd clicked Add Thread once. In this case, we have a main thread and one UI thread. (Obviously, no dialog is associated with the thread class yet, so looking at the results in Microsoft Spy++ will show only the new threads, not any associated controls.)

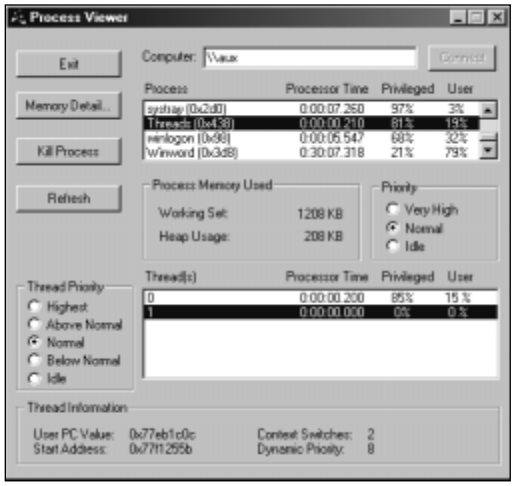

Figure 6-4: The Process Viewer shows that the Threads application is already capable of creating new threads.

The Process Viewer shows that the thread doesn't consume any time in Figure 6-4. That's because the thread isn't doing anything. Once we add a dialog to the thread's InitInstance() method, the Process Viewer will show that the thread uses a certain amount of Privileged and User time.

Adding a Thread Dialog

A UI thread always contains some type of user-interface element. You can create this element in a lot of ways, but by far the simplest is to use a standard resource. So, the first thing we'll do in this section is create a new dialog that the CUIThread class can display using the following procedure.

Try This

1. Right-click the Dialog folder on the ResourceView tab of the Workspace window, and choose Insert Dialog from the context menu. You'll see a new dialog added to the application.

2. Right-click the new dialog and choose Properties from the context menu. You'll see the Dialog Properties dialog.
3. Change the ID of the dialog to IDD_NEW_THREAD_DIALOG, then close the Dialog Properties dialog. The dialog is now ready for display, but the CUIThread class will still need a dialog class for access purposes. (We won't do a lot with this dialog because it's there for display purposes only.)
4. Ctrl+Double-click the new dialog. You'll see an Adding a Class dialog like the one shown in Figure 6-5. In most cases, you'll create a new class for a dialog. About the only time you would choose to add a dialog to an existing class is if that class natively supports dialogs and doesn't have one assigned to it. For example, if you had previously created a CDialog class derivative and hadn't assigned a dialog to it, you could use this method to do so.

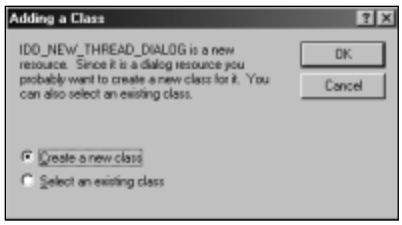
Adding a Class
IDD_NEW_THREAD_DIALOG is a new resource. Since it is a dialog resource you probably want to create a new class for it. You can also select an existing class.
OK
Cancel
Create a new class
Select an existing class

Figure 6-5: The Adding a Class dialog is where you choose to create a new class for a dialog, or assign it to an existing class.

5. Choose Create a new class, then click OK. You'll see a New Class dialog like the one shown in Figure 6-6. Notice that Visual C++ has already chosen the CDialog class for you in the Base class field. Also notice that this version of the New Class dialog doesn't include the Class type field shown in Figure 6-3. That's because you're creating an MFC class, not a general class that could be of the MFC Class type.

Figure 6-6: The New Class dialog allows you to define the characteristics of a class.

6. Type **CNewThreadDlg** in the Name field, then click OK. (Normally, you'd also need to make a choice about adding automation support to the dialog, but it's not required in this case.) Visual C++ will add the new class for you.
7. Click OK to close the MFC ClassWizard dialog.

It's time to add the new dialog to the CUIThread class. The first thing we'll need to do is add two entries to the UIThread.H file. Second, you'll need to create a variable the class can use to access the dialog. Both of these additions appear in bold type in Listing 6-2.

Listing 6-2: **New Thread Dialog Access Entries in UIThread.H**

```
// Include support for the New Thread Dialog
#include "NewThreadDlg.H"


/////////////////////////////////////////////////////////////////////////////
// CUIThread thread

class CUIThread : public CWinThread
{
    DECLARE_DYNCREATE(CUIThread)
protected:
    CUIThread();           // protected constructor used by dynamic creation

// Attributes
public:

// Operations
public:

// Public Variables
    CNewThreadDlg    dlgNewThread;     // New Thread Dialog access
```

The InitInstance( ) method is what controls the starting point for each new thread that the application will create. We'll simply place the dialog display code in there. Listing 6-3 shows the code you'll need to use in this case.

Listing 6-3: **InitInstance( ) Method Source Code**

```
BOOL CUIThread::InitInstance()
{
    // Display the thread dialog.
    m_pMainWnd = &dlgNewThread;
```

```
    dlgNewThread.DoModal();

    // Returning TRUE will destroy the thread.
    return TRUE;
}
```

All that this code does is assign the dialog as the main window for the thread. This is a very important thing to do because otherwise, you'll only be able to create a single new thread. (Displaying the dialog modally without assigning it as the main window prevents you from going back to the initial dialog and clicking Add Thread.) Once the dialog is assigned as the main window for the thread, we display it modally. When the user clicks OK, the dialog call returns and the thread returns true. This act destroys the thread.

Creating a Worker Thread

At this point, you could compile the application and see the effects of a UI thread. Of course, the thread dialog isn't doing anything at this point. That's where the worker thread we'll create in this section comes into play. We'll perform two tasks required to create a worker thread that sleeps for five seconds, and then ends.

We'll also look at some thread synchronization issues in this section. For example, you'll learn about the importance of waiting for any worker threads to end before terminating a main thread (in this case, the UI thread we created in the previous section). We'll modify the functionality of the OK button to ensure that the worker thread has actually ended before we close the dialog. This ensures that the dialog doesn't create any memory leaks.

The Sleep() method shown in this section can be used to replace a timer when working with threads. The thread will sleep for a certain amount of time, wake itself up, then perform whatever work it needs to do before going back to sleep.

The first thing we'll need to do is create the thread. I've chosen to create the thread as part of the process of creating the dialog, which means we'll have to change the default functionality of the OnInitDialog( ) method. Since our application doesn't include the handler for the CNewThreadDialog::OnInitDialog( ) method, we'll need to add it. To do this, open the MFC ClassWizard, choose the Message Maps tab, then add a handler for the WM_INITDIALOG message of the CNewThreadDlg class. Figure 6-7 shows what your MFC ClassWizard dialog should look like. Click Edit Code to get to the new method definition. Listing 6-4 shows the code you'll need to add for this example.

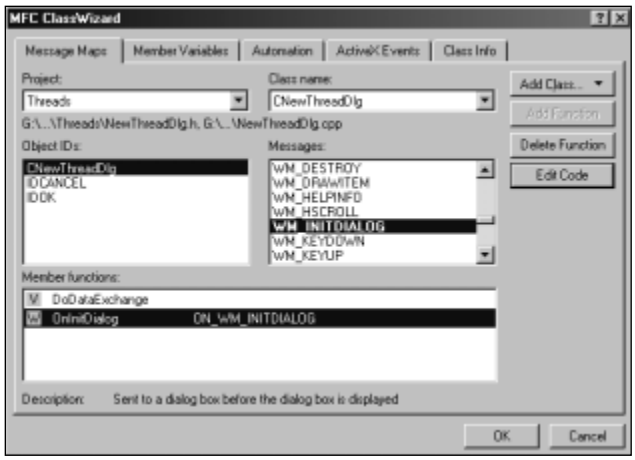

Figure 6-7: You'll need to add a WM_INITDIALOG message handler to the CNewThreadDlg class.

Listing 6-4: **OnInitDialog() Method Source Code**

```
BOOL CNewThreadDlg::OnInitDialog()
{
    // Perform the default action.
    CDialog::OnInitDialog();

    // Create a worker thread.
    pThread = AfxBeginThread(SleepABit,
        NULL,
        THREAD_PRIORITY_BELOW_NORMAL,
        0,
        0,
        NULL);

    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}
```

The default action and return code are both provided by the MFC ClassWizard, so the only thing you need to add is the AfxBeginThread( ) function. You'll need to define the pThread variable in the public section of the header file as shown here:

```
    CWinThread    *pThread;     // New thread pointer.
```

There are six arguments you pass to AfxBeginThread( ). The following list describes each argument and what you'd use it for.

Function Name The first argument is the name of the worker thread function, which can't be a member of the current class. (We'll discuss how this works in a bit.)

Function Parameter The second argument is a parameter that you can pass to the worker thread function. It's of type LPVOID, so you'll need to typecast it within the worker thread to the desired variable type. Many thread writers use this parameter to pass a BOOL that tells the worker thread to terminate. However, you can use the parameter for any purpose and can even pass multiple parameters in the form of a structure.

Thread Priority This is the default thread priority. It's actually a relative measure of the thread's priority when compared to the parent, and not an absolute priority level. The example code shows the worker thread starting at a priority level one point below the parent thread. A value of THREAD_PRIORITY_NORMAL would start the thread at the same priority as the parent.

Stack Size Providing a value of 0 means that you want to use the default stack size, which is 1MB maximum in most cases. Windows doesn't actually allocate 1MB of stack space; it allocates only what the application actually needs. The only time you'd want to specify a stack size is if you wanted to ensure recursive code or other code elements didn't overflow the stack. Obviously, you'd also need to provide a value here in those rare situations in which a thread might need more than 1MB of stack space.

Creation Flags At present, there are only two creation flags. A value of 0 tells Windows to start the thread immediately. You can also supply a value of CREATE_SUSPENDED, which means that you'd have to resume the thread before it would run the first time. The Suspend( ) method will allow you to stop a thread after its created, whereas the Resume( ) method will start it, so you still have control over the thread later using the thread object pointer. This is one of the reasons we saved the thread object pointer as a public variable.

Security	Providing a NULL value for this argument starts the thread with the same security attributes as the parent thread. You'd need to provide a pointer to a SECURITY_ATTRIBUTES structure if you wanted to change any of the thread's security levels. We'll look at the SECURITY_ATTRIBUTES structure in Chapters 21 and 22 of the book.

Now that we have a thread, we'll need to do some synchronization to ensure the UI thread doesn't try to end before the worker thread does. That means adding some code to the OnOK( ) method of the CNewThreadDlg class. Adding the method is easy, just Ctrl-double-click the OK button of the IDD_NEW_THREAD_DIALOG dialog. Listing 6-5 contains the code you'll need to add for the OnOK( ) method.

Listing 6-5: **The OnOK( ) Event Handler Source Code**

```
void CNewThreadDlg::OnOk()
{
    // Wait until the thread has terminated.
    WaitForSingleObject(pThread->m_hThread, INFINITE);

    // Perform the default OnOK() processing.
    CDialog::OnOK();
}
```

The WaitForSingleObject( ) function allows us to detect when the thread has finished executing. The INFINITE argument tells Windows that we'll wait forever for the worker thread to end. There is a potential problem with using this particular argument. Unless you're absolutely certain that the thread will actually end, the application could appear to freeze if the thread doesn't end for some reason. In many cases, it's actually better to provide a value in milliseconds for the second argument. If the thread doesn't finish executing in a given amount of time, you can always add code that allows your application to detect the problem, display an error message for the user, and end gracefully.

Calling CDialog::OnOK( ) allows parent class to complete any handling for the event. It's important to ensure that the event gets handled. Obviously, unless you have some special processing to perform, it's easier to allow the parent class to handle the default processing needs.

It's finally time to add the worker thread code. You'll need to add a function declaration before the class definition in NewThreadDlg.H like the one shown here:

```
// Add a declaration for our worker thread.
UINT SleepABit(LPVOID pParam);
```

Make absolutely certain that you add the worker thread function declaration *before* the class declaration or the example won't compile properly. The worker thread function is not part of the CNewThreadDlg class, and therefore won't appear in the ClassView tab of the Workspace window. In addition, a worker thread must always follow the format shown in the declaration. It must return a UINT and accept a LPVOID parameter value, even if the parameter isn't used within the worker thread function.

The SleepABit( ) function itself is relatively simple. Listing 6-6 shows the source code for this function.

Listing 6-6: **The SleepABit( ) Worker Thread Source Code**

```
UINT SleepABit(LPVOID pParam)
{
    // Sleep for 5 seconds.
    Sleep(5000);

    // End the thread
    return 0;
}
```

As you can see, the source for the worker thread is extremely simple. All we do is tell the thread to sleep for 5,000 milliseconds, then end.

Testing the Threads Application

Compile the Threads application so that you can see how the various threads work together. There are several ways that you can view this example. First, start the application, click Add Thread, then immediately click OK on the resulting Dialog dialog. The dialog won't actually disappear for five seconds. What you're seeing is the result of the combination of the Sleep( ) and WaitForSingleObject( ) functions. The worker thread sleeps for five seconds after it get created. Because the worker thread is created as part of creating the UI thread, there isn't any way to close the dialog for five seconds after you start it.

Let's look at this from a different perspective. Open the Process Viewer. Find the Threads application in the Process list, click Add Thread on the Threads Application, then Refresh on the Process Viewer. You'll see three threads as shown in Figure 6-8. The first thread is the main application thread, which will stay even after you close any created thread dialogs. The second is the UI thread, which will remain until you click OK on the Dialog dialog. Finally, the third thread is the worker thread—it'll cease to exist in five seconds. Wait five seconds, click Refresh, and you'll see that the worker thread has indeed disappeared from the thread list.

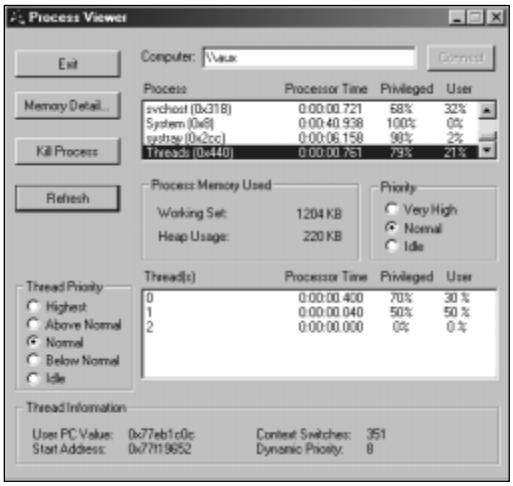

Figure 6-8: The Process Viewer is helpful for showing which threads are executing and how much processor time they're using.

Notice that the worker thread shown in Figure 6-8 isn't using any processor time because it's in sleep mode. In addition, if you click on the worker thread entry, you'll see that Thread Priority field changes to Below Normal and that the actual worker thread priority is at least one below the main thread's priority.

It's interesting to check things like process memory usage each time you start a new thread. You'll find that each thread consumes about 24Kof heap when first started and 12K of heap once the worker thread has ended. The 12K heap represents the amount that Windows allocates for housekeeping as well as the thread itself. You can get a more detailed account of memory usage by clicking the Memory Detail... button in Process Viewer. You may also want to see the effects of changing the process priority on thread creation. Even though all of the priority levels will rise or fall, the worker thread will normally remain exactly one priority level below the main thread — even though the UI thread is at a higher priority than the main thread because it has focus.

Spy++ provides another chance to see the effects of the two thread types in this application. Figure 6-9 shows a typical Processes window for the Threads application that includes both the UI and worker thread. Notice that the UI thread (as its name suggests) contains a list of user interface components. Although the Process Viewer is very helpful in determining memory usage and priority levels, you'll find that Spy++ is very helpful in tracking thread windows and message traffic. You really need both applications when troubleshooting problem applications that use threads.

Figure 6-9: Spy++ will help you track both windows and messages created by the threads associated with an application.

Writing a Basic DLL and Using It in an Application

DLLs have always been one of the mainstays of the Windows programmer. The reasons for using them are easy to understand. A DLL allows you to place commonly used code within an easy-to-access module that you don't need to recompile every time you create a new version of your application. In addition, DLLs allow you to modularize your application, making it easier to troubleshoot (especially since you'll have already debugged the code within the DLL). Using DLLs can also be very memory efficient. The DLL gets loaded only when and if you actually need it (provided you code your application correctly), making the initial memory footprint for an application smaller.

It's no surprise that some developers want to combine the good features of threads with DLLs. Placing your common thread code within a DLL makes sense for the same reasons that you would use this technique with any application code. In addition, there isn't any reason why a DLL can't use internal threads as well — threads that are created for the sole purpose of making the DLL more efficient and responsive.

The use of threads within DLLs has a few caveats. The following list will provide you with some of the more common problems that you'll experience.

Thread Safety

When you place a thread within an application, you're fairly certain about the application that will access it and how that access will occur. Any number of applications could access a thread within a DLL and you need to take that into account. Thread safety becomes more critical, in this case, so you must ensure that data access is checked and critical areas are protected.

Exporting Requirements	You'll find that worker threads are much easier to place within a DLL simply because they're easier to export. In many cases, figuring out how to export complex UI thread objects proves to be more difficult and error prone.
Development/Debugging Time	The problems that you'll experience writing and debugging code for an application are exacerbated when working with DLLs. Not only do you have to take parallel processing into account, but the DLL as well. In other words, you may want to consider developing the thread code within the application environment first, then moving it to a DLL if needed. Debugging the thread code first will make it easier to determine where a problem lies when you create the DLL.

Now that we have a few preliminaries out of the way, let's look at an example of a worker thread that we can call from a DLL. The thread function will reside within a DLL. As with the previous example, the thread itself will be very simple so that you can better see the mechanics of using the thread itself. In the following sections, I discuss how to create the example program.

Creating the DLLThread DLL

Our application will actually consist of two projects. The first is a DLL that will hold the worker thread. The following procedure will get you started.

Try This

1. Create a new MFC AppWizard (dll) project named DLLThread. Make absolutely certain that you create a DLL project, not an EXE project. Visual C++ will display the MFC AppWizard - Step 1 of 1 dialog shown in Figure 6-10. As you can see, the MFC AppWizard allows you to choose from three different DLL types. The first two options produce DLLs that any application can call. The first type links the required MFC functions statically, which increases the size of the DLL in memory, but ensures that the required MFC support is bundled with the DLL. The third DLL option produces an MFC extension. Only MFC applications can call this type of DLL, making it an unlikely choice for general application development.

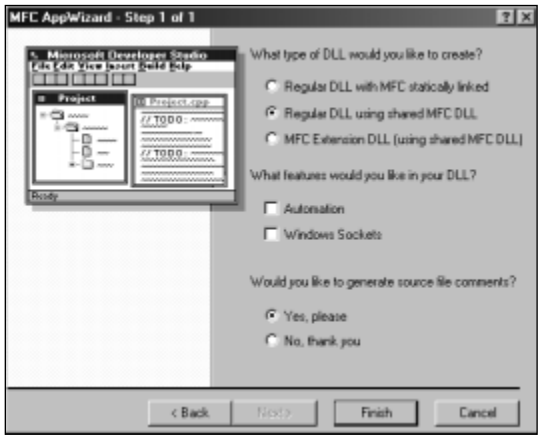
MFC AppWizard - Step 1 of 1
What type of DLL would you like to create?
Regular DLL with MFC statically linked
Regular DLL using shared MFC DLL
MFC Extension DLL (using shared MFC DLL)
What features would you like in your DLL?
Automation
Windows Sockets
Would you like to generate source file comments?
Yes, please
No, thank you
< Back
Next >
Finish
Cancel

Figure 6-10: The MFC AppWizard - Step 1 of 1 dialog allows you to choose the type of DLL to create.

2. Choose the Regular DLL using shared MFC DLL option, then click Finish. Visual C++ will display the New Project Information dialog shown in Figure 6-11.

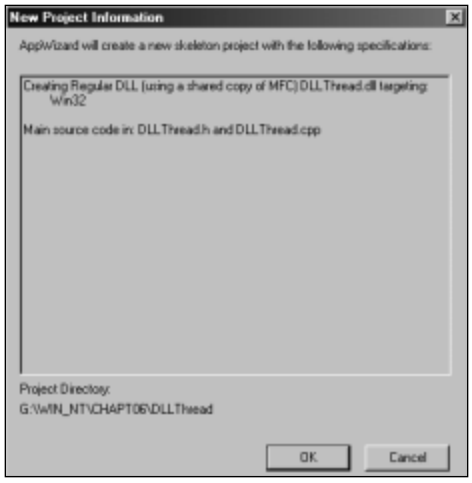
New Project Information
AppWizard will create a new skeleton project with the following specifications:
Creating Regular DLL (using a shared copy of MFC) DLLThread.dll targeting:
Win32
Main source code in: DLLThread.h and DLLThread.cpp
Project Directory:
G:\WIN_NT\CHAPT06\DLLThread
OK
Cancel

Figure 6-11: The New Project Information dialog shows that this is a regular DLL.

3. Click OK. Visual C++ will create the DLL shell for you.

CD-ROM Location: Chapt06\DLLThread
Program Name: DLLThread.DLL
Source Code Modules in Text: DLLThread.CPP

Adding the worker thread to the DLL is relatively easy. Listing 6-7 shows the code you'll need to add for this example.

Listing 6-7: **The DoDialog( ) Worker Thread Source Code**

```
__declspec(dllexport) UINT DoDialog(LPVOID pParam)
{
    // Required pre-processing.
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    // Display a confirmation message.
    AfxMessageBox("Thread Started", MB_OK);

    // End the thread
    return 0;
}
```

Notice that the general calling syntax for this worker thread is the same as before. The big difference is the addition of __declspec(dllexport) to the beginning of the call. This extra code tells the compiler that we want to export the DoDialog( ) function for use outside of the DLL (we'll see the effect of this extra code in a moment).

The __declspec( ) function is a Microsoft-specific extension to Visual C++, which makes the code nonportable to other environments. You can get around this problem by using the more common DEF file technique for exporting the function. However, this means maintaining extra code that the __declspec( ) function takes care of for you automatically. The only other reason to use a DEF file is if you need to control the ordinal order of the exported functions. Using __declspec( ) means that you have to reference a DLL function by name rather than ordinal number.

Let's look at the DoDialog( ) function. The first call is to the AFX_MANAGE_STATE( ) macro, which performs a context switch from the main application to the DLL. This macro ensures that Windows looks for resources within your DLL, rather than the main application. You absolutely have to include this macro if you plan to use any MFC calls or local resources, like icons or bitmaps in most functions. There are, however, exceptions to the rule. For example, Windows automatically performs a context when you call the InitInstance( ) method of a class.

The remaining code is quite simple. The DoDialog( ) function displays a standard message box. After the user closes the message box, the function returns a 0, which terminates the thread.

Compile the DLL. Now, find the DLL file in either the Debug or Release folder of the project folder using Windows NT Explorer. Right-click the file and choose Quick View from the context menu. Scroll down the list of entries in the Quick View window until you find the Export Table. Your Quick View window should look similar to the one shown in Figure 6-12. Notice that the "decorated" (or mangled) version of the DoDialog( ) function appears in the list of exported items. It's always helpful to check this list to ensure the functions you'll need outside the DLL are actually present.

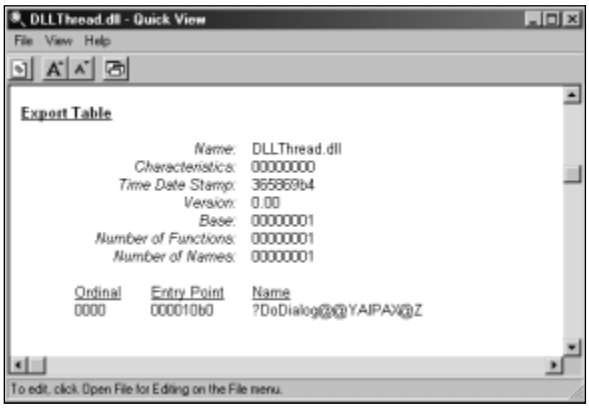

Figure 6-12: The Quick View utility will show you a list of exported function names for your DLL.

Creating the DLLTest Application

It's time to create an application that can use the DLL that we just created. We'll use a dialog-based application that looks very similar to the one we created earlier in the chapter.

Try This

1. Create a new MFC AppWizard (exe) project with the name DLLTest. You'll see the MFC AppWizard - Step 1 dialog.
2. Choose the Dialog-based option, then click Finish. You'll see a New Project Information dialog like the one shown in Figure 6-13.

Figure 6-13: The New Project Information dialog for the DLLTest project.

3. Click OK. Visual C++ will create the new project for you, then display the main application dialog in the Design window.
4. Remove the Static Text control and add a Button control.
5. Open the Push Button Properties dialog. Change the ID to IDC_ADD_THREAD and the Caption to Add Thread. Your dialog should look similar to the one shown in Figure 6-2. (There will be some differences like the application name on the title bar that you don't need to worry about.)
6. Ctrl+Double-click IDC_ADD_THREAD. Visual C++ will add a new procedure for the button to the application. Before we can add any code for the new procedure, we also need to make changes to the project settings so that Visual C++ will know where to access the DLLThread.LIB file (which contains the information required to use the DLLThread DLL that we created in the previous section).
7. Use the Project | Settings... command to display the Project Settings dialog.
8. Choose the Link tab of the Project Settings dialog. You should see a list of link settings like the ones shown in Figure 6-14.
9. Type **..\DLLThread\Release\DLLThread.LIB** in the Object/library modules field, you may need to change this entry depending on where the DLLThread.LIB file is located on your machine.
10. Click OK. Visual C++ will change the project settings to include a location for the DLLThread.LIB file.

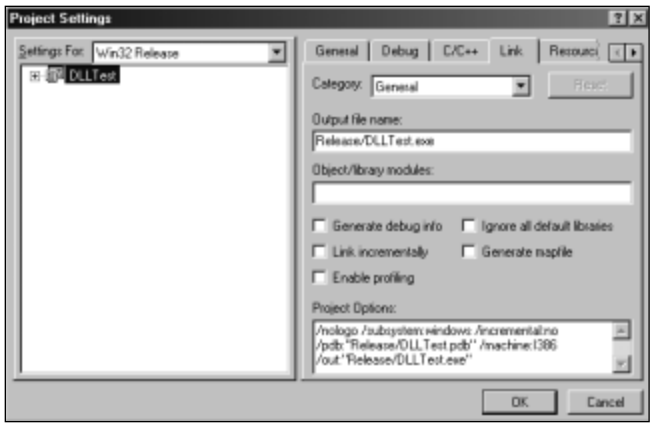

Figure 6-14: The Link tab of the Project Settings dialog will allow you to add the DLLThread DLL to the project.

We'll need to make some changes to the DLLTest.H file as well. The first will add an import entry for the DoDialog( ) function. The second is a CWinThread object pointer, like the one we used in the previous example. Both of these changes are shown in bold type in Listing 6-8.

Listing 6-8: **DLLTest Header Changes**

```
// DLLTestDlg.h : header file
//

// Import the DoDialog() function.
__declspec(dllimport) UINT DoDialog(LPVOID pParam);

#if !defined(AFX_DLLTESTDLG_H__666FF698_823F_11D2_9D78_00A02456674B__INCLUDED_)
   #define
   AFX_DLLTESTDLG_H__666FF698_823F_11D2_9D78_00A02456674B__INCLUDE
   D_
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

/////////////////////////////////////////////////////////////////////////////
// CDLLTestDlg dialog

class CDLLTestDlg : public CDialog
{
// Construction
```

```
public:
    CDLLTestDlg(CWnd* pParent = NULL);    // standard constructor

    CWinThread    *pThread;    // New thread pointer.
```

At this point, the main application dialog is ready to go. All we need to do is add some code to make the Add Thread button functional. Listing 6-9 shows the code you'll need to add.

Listing 6-9: **The OnOK Event Handler Source Code**

```
void CDLLTestDlg::OnAddThread()
{
    // Create a worker thread.
    pThread = AfxBeginThread(DoDialog,
        NULL,
        THREAD_PRIORITY_NORMAL,
        0,
        0,
        NULL);
}
```

As you can see, all that this code does is create a thread using the imported DoDialog( ) function. In sum, calling the DLL version of the worker thread is not all that different from using a worker thread within the application itself.

You must copy the DLLThread.DLL file into the DLLTest application folder before you attempt to run the application. Otherwise, Windows will complain that it can't find the DLL file.

If you ran that application at this point, you'd see many of the same things we did for the previous example. The process viewer would show a new thread every time you click the Add Thread button. Using a DLL does require a little more memory — as you'll see when using Process Viewer. Looking at the application with Spy++ will show the main thread and associated window, along with any worker threads that you create. However, because we're creating a message box within the worker thread this time, you'll also see a dialog associated with each worker thread.

Summary

Learning to work with threads will allow you to create applications that are both efficient and responsive to user input. An understanding of threads is also essential when creating some types of complex applications — especially those that rely on remote connections or asynchronous communication. Here are some additional conclusions you can draw from the contents of this chapter:

- Threads are useful in different ways for applications, DLLs, and system services.
- There are two kinds of threads supported by MFC: worker and UI.
- Worker threads are created using a standard function format and normally don't include any user interface elements.
- UI threads are derived from the CWinThread class and normally provide some type of user-interface feature.
- The only CWinThread method that you must override when working with a UI thread is InitInstance( ).
- Critical sections are created using a CCriticalSection object.
- You use critical sections to protect variables and objects from access by more than one thread.
- MFC libraries are thread safe at the class level, not the object level.
- You can use a variety of methods to create thread-safe libraries of your own, including the use of critical sections.
- The default-thread priority is the same as the base application when it has focus, but you can set it lower or higher during the creation process.
- Threads normally start running, but you can also create them in a suspended state to perform additional processing before allowing the thread to resume.
- Though the default stack size for a thread is 1MB, Windows allocates only the amount of memory that the thread actually needs.

✦ ✦ ✦